Gelato @ UNSW

Performance and Scalability on Itanium

www.gelato.unsw.edu.au

peterc@gelato.unsw.edu.au

NFS Performance and Scalability

Work-In-Progress

Shehjar Tikoo and **Peter Chubb**

April 2007

http://www.gelato.unsw.edu.au/IA64wiki/NFSBenchmarking

From NFS-Linux mailing list:

```
[NFS] poor nfs performance
[NFS] Performance expectations of NFS
```

Little useful response; NFS still slower than it should be

There have been many complaints on the Linux Kernel Mailing list and the Linux NFS mailing list about NFS performance. Sometimes (as in the first case above, where 2000 NFS mounts were taking over 30 minutes to complete on Linux clients, versus less than a minute on Solaris) the problem can be solved. Other times, (the second case, where copying from NFS server one to NFS server two is faster than copying from local disk to NFS server, or from NFS server to local disk, and two orders of magnitude slower than a local disk to local disk copy). But there is little concerted effort to ensure that NFS performance is and remains good. In fact, it appears that the NFS maintainers are concentrating on NFSv4, rather than the more widely deployed NFS v3.

Why NFS?

- Used extensively — e.g., for home directories
- A standard — Interoperable with other OS
- could perform reasonably.

NFS is widely deployed for home directory services, even if other distributed file systems are used for the main cluster file system. It provides close-to normal UNIX file system semantics (unlike SMB), and there's no reason it shouldn't perform fairly well — in fact it does perform well on Solaris and Irix, and reasonably well on Linux on single processor x86 machines. But it appears to perform much worse than it should on IA64 and SMP.

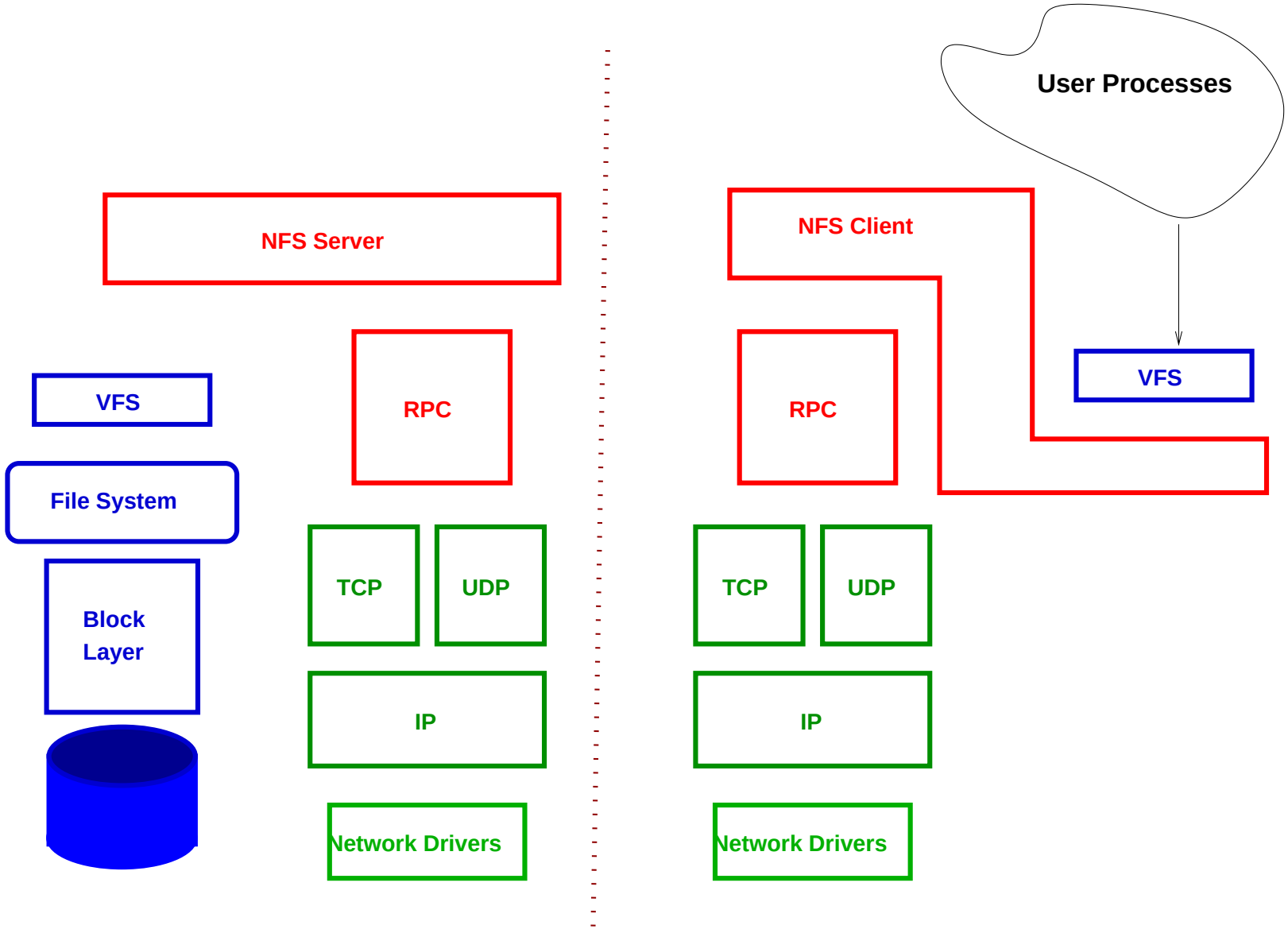
Our Aims

- To Measure NFS server and client performance
- To Understand the NFS parameters
- To Improve NFS performance

The first step is to measure and understand the performance as it currently is. What are the limits to performance? At the server? At the client? Where are the delays, the bottlenecks, the points of contention?

To answer these questions we need repeatable benchmarks that exercise each part of the NFS stack. We can then profile the kernel, and find where the problems are. We can also vary the various NFS and network parameters and see how they affect performance.

NFS stack


User Processes
NFS Server
NFS Client
VFS
RPC
RPC
VFS
File System
Block Layer
TCP
UDP
TCP
UDP
IP
IP
Network Drivers
Network Drivers

NFS starts from userspace on the client; requests made by user processes are translated into VFS requests by the kernel. Some of these requests can be satisfied from the page-cache; others may trigger VFS read-ahead on the client machine. When actual I/O is necessary, the NFS client performs remote-procedure calls to the NFS server process on the NFS server machine. RPC involves marshalling arguments, and converting to network byte ordering, then passing the resulting packets to either a TCP or UDP network stack. Often packets (or replies) are larger than the network physical layer maximum packet size, so the upper networking layers fragment the packets and add appropriate headers before passing down to the network drivers.

This all has to be undone on the server side. Eventually, the RPC becomes a local function call to the NFS daemon. It will then call down into the VFS layer to do the work; again, the request could be satis-

fied from the NFS server's pagecache, and the request could trigger readahead.

The RPC reply process is very similar to the call.

Key performance points are:

- Server file system performance:
  - The exported file system type.
  - The elevator algorithm on the server.
  - Fragmentation on the disk.
- Network performance and latency
  - Ethernet Frame size

  - Whether payloads can be made page-aligned
  - Byte ordering — if the machines are big-endian no conversion will be necessary

- Page cache performance
- Read-ahead on client and server
- When the client calls `sync`. `Sync` requires outstanding write requests to be flushed to disk.

Methodology

- Benchmark Client and Server separately.
- Write new Open Source NFS server stress test.
- Use standard FS benchmarks for client.
- Also (eventually) use full-system benchmarks like AIM7.

In order to avoid confusion, we decided to characterise server performance first. When we know how the server behaves, we can use it for client benchmarking using standard filesystem benchmarks such as iozone and fsstress.

The ‘standard’ way to benchmark an NFS server is with SpecSFS. But SpecSFS is a proprietary benchmark. So we decided to write our own, Open Source version, that would be freely distributable.

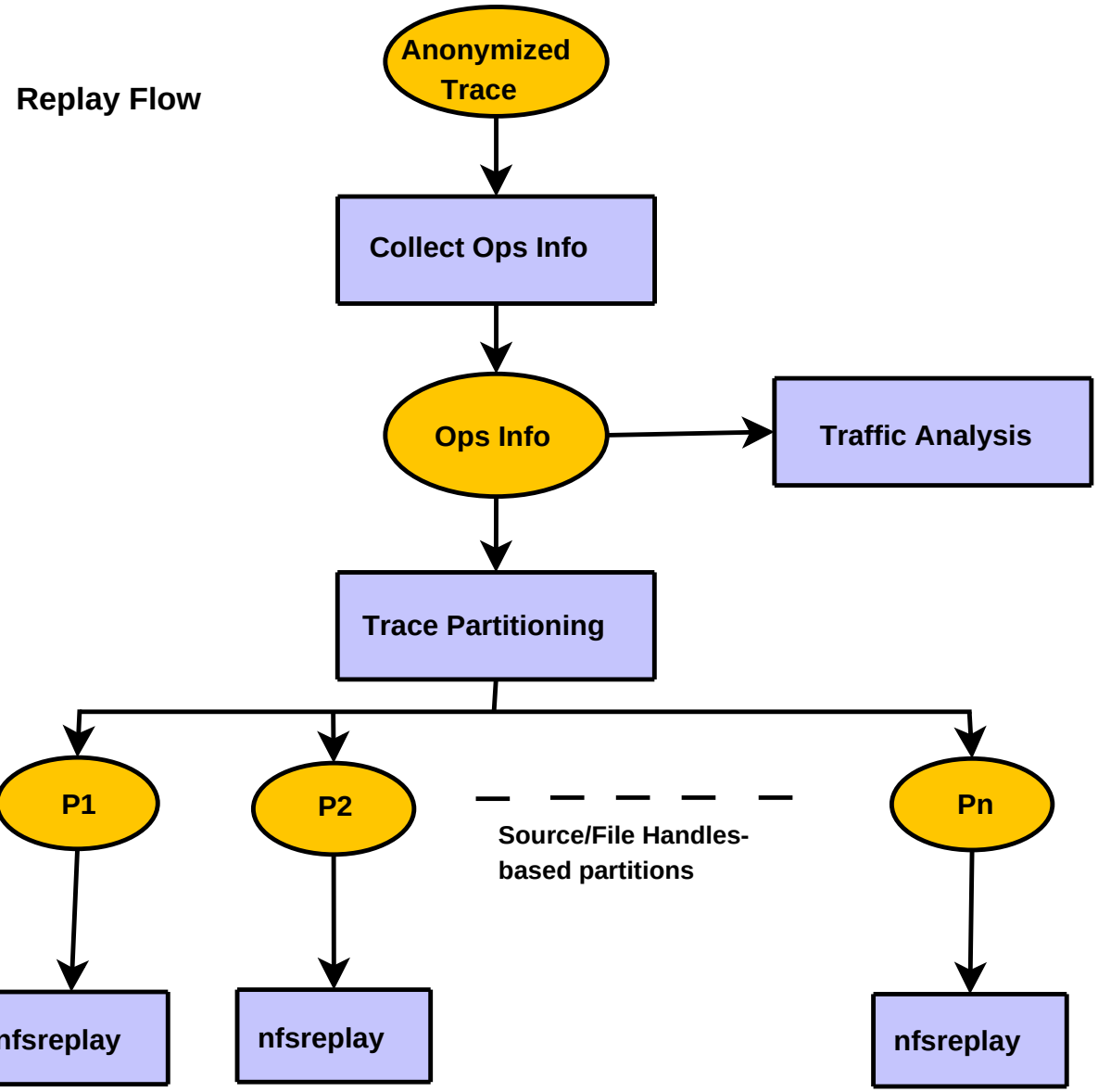
Testing the Server

- Use real workloads.
- Avoid using NFS client.
- Use multiple clients to reduce client-side load.

To remove one of the variables, we want to test the server without using a (possibly performance limited) NFS client. We also want to use real workloads where possible.

That means collecting traces from people doing real work. But companies are (rightly) paranoid about information about file contents leaking from their premises. So the first step is to write an anonymiser that plugs into wireshark and replaces details of file contents, names, IP addresses, etc., with tokens. The resulting trace can be analysed for patterns, and to reproduce a filesystem that can be used as a target for tests.

Replaying NFS traffic



Once the filesystem has been created and exported from an NFS server, we can replay the trace at a higher speed, by splitting independent operations and passing them to multiple client servers. The devil, however, is in the details.

Work In Progress

- Binary NFS Traffic Anonymizer — **Done**
- `nfsreplay`, NFS Traffic Replayer — **Ongoing**
- Benchmarking — **Ongoing**

Shehjar's finished the anonymiser. Although it has not been accepted into the wireshark system itself, the changes to allow its integration have been, so building wireshark with an anonymiser is almost trivial.

The replayer is more work. To keep our hand in in the mean time, we've started a bit of client side benchmarking. And have found some interesting results (see below!)

NFS Traffic Anonymizer

- Anonymizes packets in binary/pcap format
- Developed as a plug-in for TShark
- Details at

  http://www.gelato.unsw.edu.au/IA64wiki/NFSTrafficAnonymizer

nfsreplay

- Replays anonymized NFS traffic
- Sub-projects
  - Asynchronous RPC
    - **Asynchronous** and **non-blocking** RPC calls allow scalable replay
    - Current glibc RPC does not provide this
    - Being developed as a transport layer handler — Can be plugged into glibc

To act as an efficient userspace NFS client, we needed to create a system to allow asynchronous non-blocking RPC (so we could initiate several at specific time intervals without waiting for replies). The RPC code in glibc doesn’t allow this, so we’re writing our own.

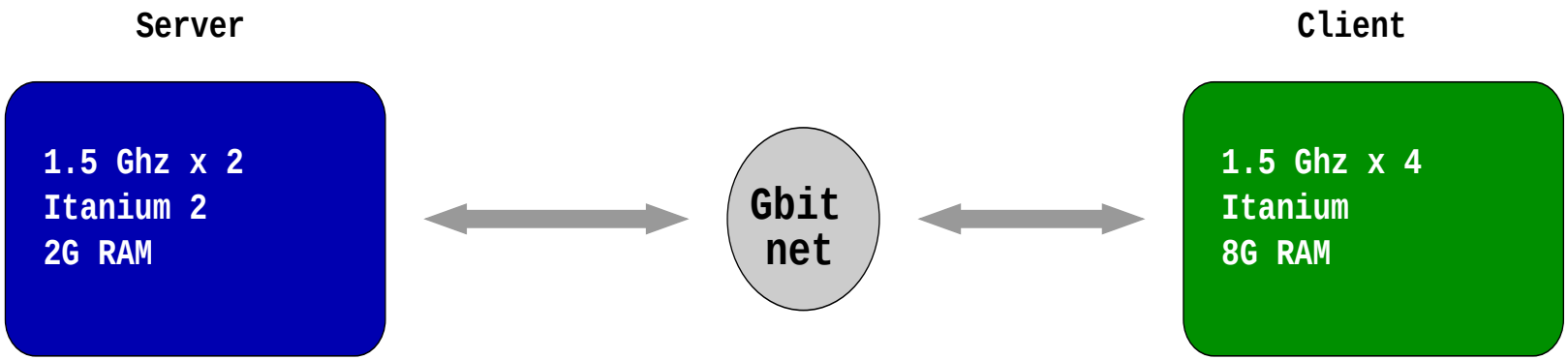
nfsreplay

- Sub-projects
  - Client NFS library
    - Currently no library exists for NFS client ops from user space
    - Builds on the Asynchronous RPC lib
- http://www.gelato.unsw.edu.au/IA64wiki/nfsreplay

On top of the RPC layer, we need an NFS library. While there exists a user-space NFS server, there is not (yet) a user-space NFS client. So we're writing one.

Benchmarking

Setup



To benchmark the NFS client, we attached it to a server. In these tests, the client is 4-way Altix 350; the server a 2-way RX2600 with a separate SCSI 360 disk exported for NFS.

It’s surprisingly frequently the case that the NFS server is underpowered relative to the clients it serves.

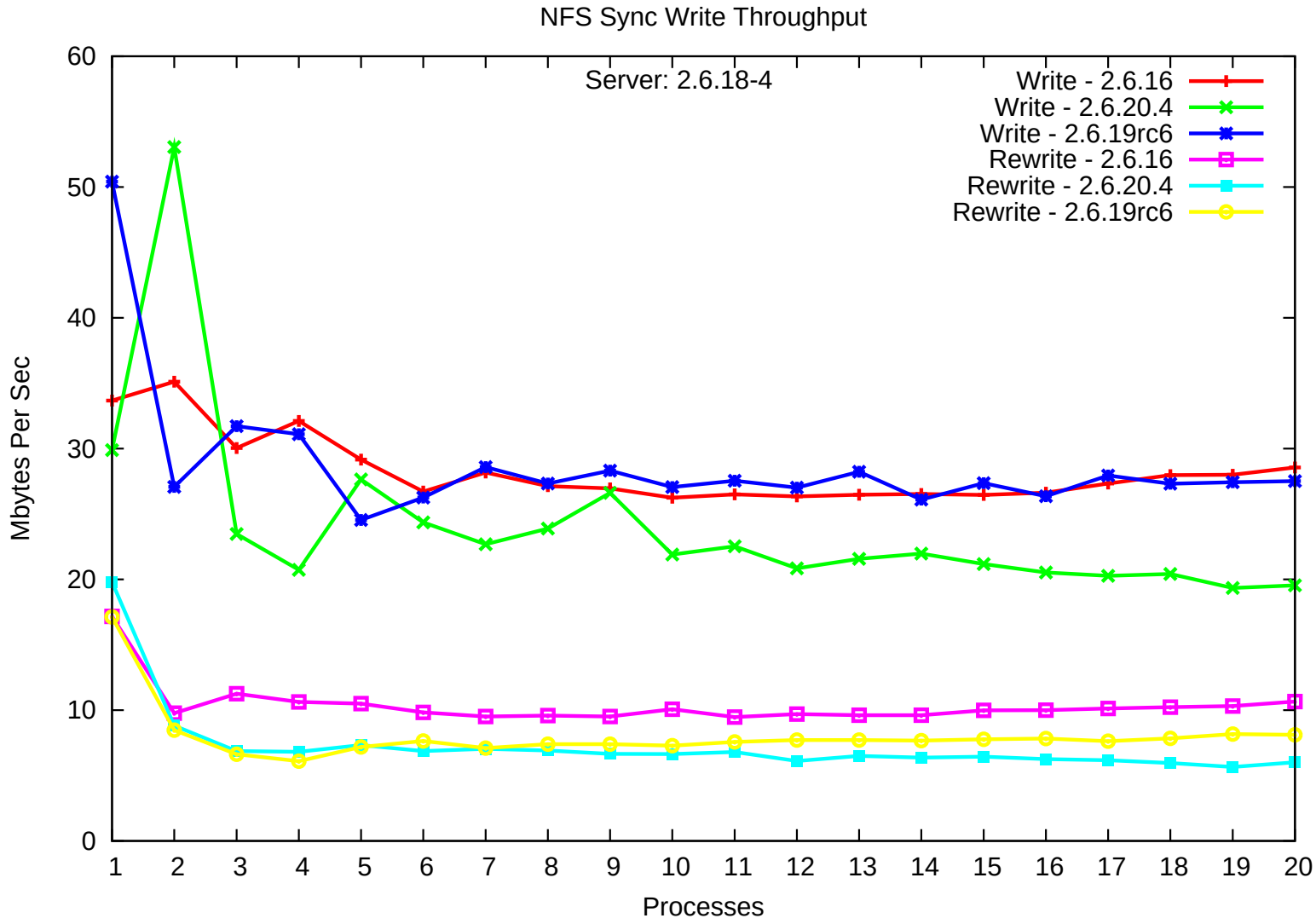
Benchmarking

Iozone Tests

- Throughput and Client CPU Utilization measurement
- Writes/Re-writes and Reads
- iozone processes in range 1 to 20
- Per process read/write-100M records, File size 2G
- Kernels tested: 2.6.16, 2.6.19-rc6, 2.6.20.4

Iozone is from http://www.iozone.org. It's fairly configurable; the tests reported here create 2G files for each process, and perform I/O in 100M chunks. The number of processes varies from 1 to 20, and the aggregate throughput is measured and graphed. The total CPU utilisation during each run is measured at the same time.

NFS Synchronous Writes and Rewrites



- Rewrite perf is bad for all three versions. This could partly be because of lack of memory on the server; it's also because RPC payloads are not page-aligned on the server.

- Throughput holds up fairly well under high loads. But it's still lower than expected.

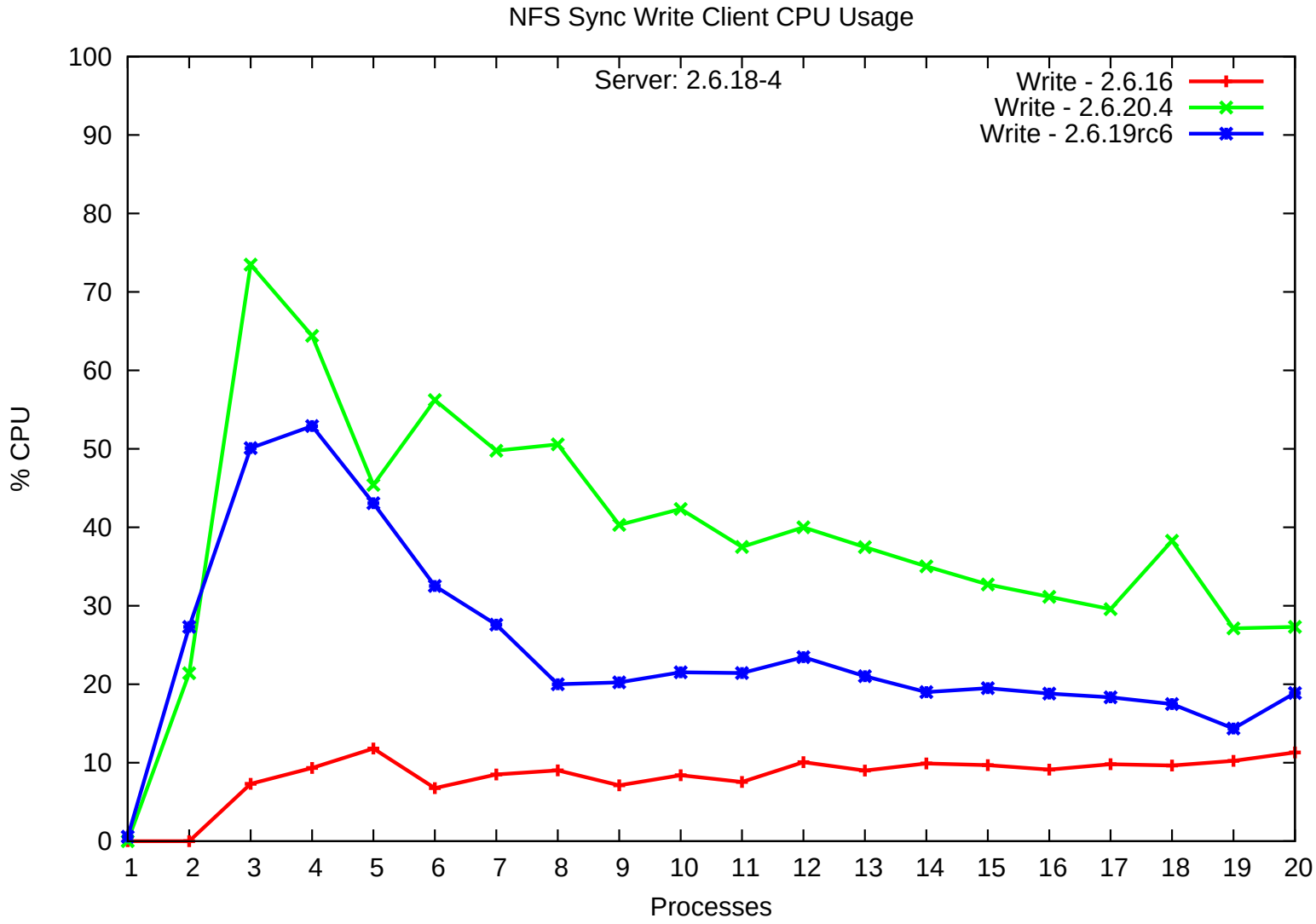
- Low and drastically falling throughput when process count is small.

- Read performance is lower than writes, and falls with increasing parallel process count. This is partly because of lack of page-cache memory on the server (normally you'd expect throughput to increase with processes until something reaches saturation).

The rest of the poor performance is unexplained, but I suspect something is serialising requests when that’s unnecessary.

- 2.6.20.4 throughput is consistently lower beyond 10 processes.

NFS Client CPU usage

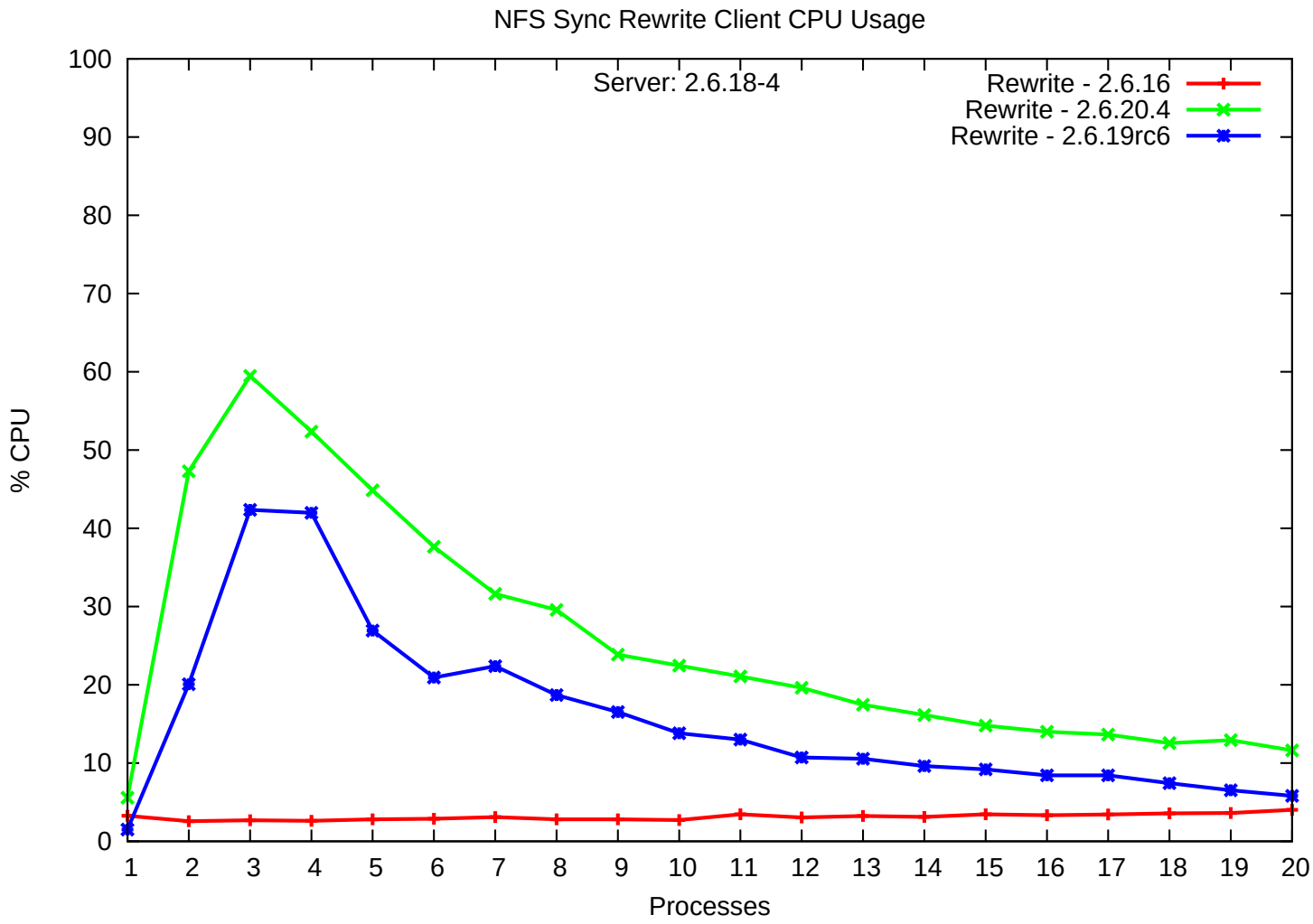


Ideally, the throughput should increase with number of concurrent I/O processes before getting bounded by network and disk.

Here, high CPU usage seems to pull it down instead.

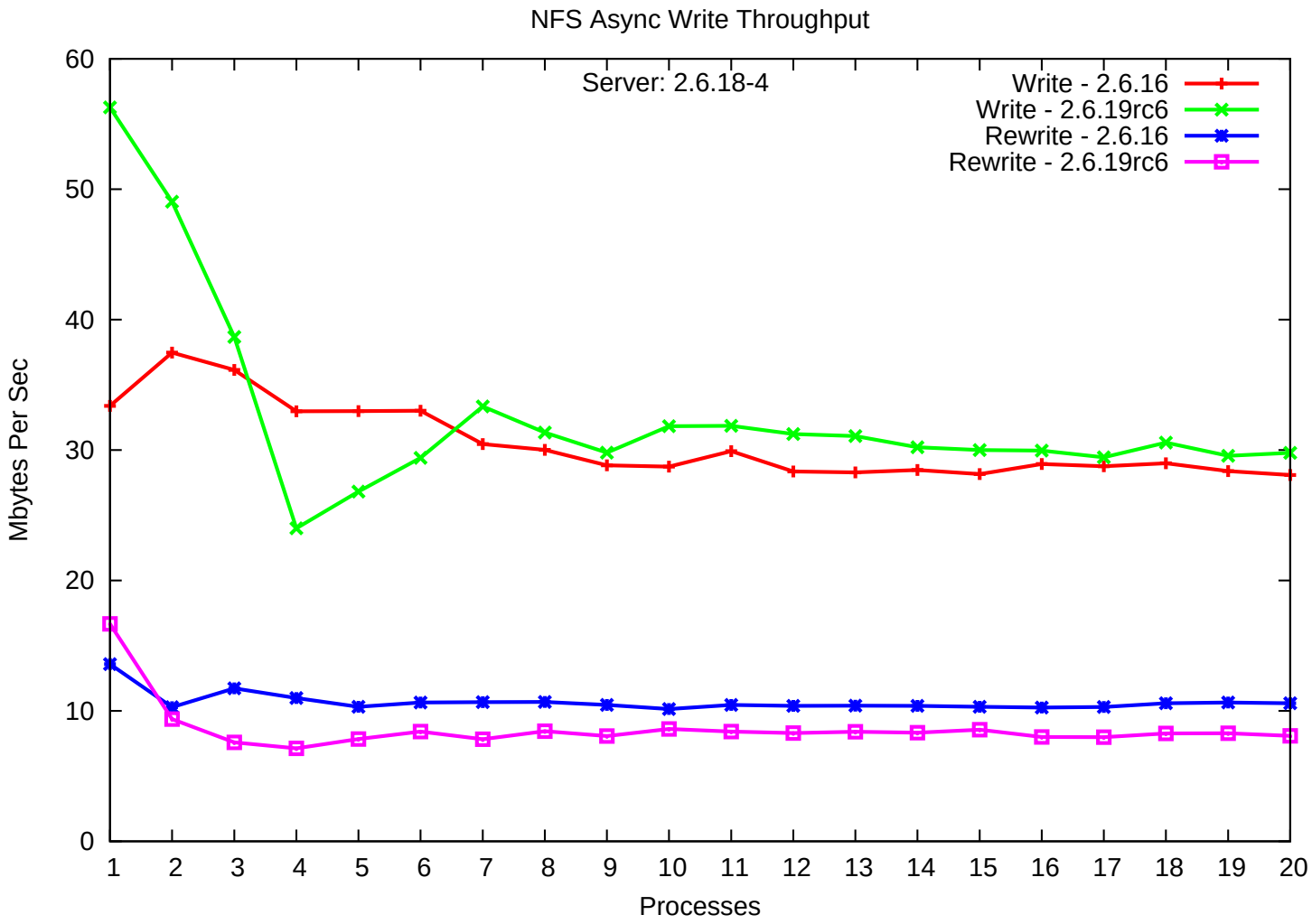
Note the major regression since 2.6.16 in CPU usage.

NFS Rewrite CPU usage



Rewrite performance is terrible; CPU usage is high for recent Linux kernels, but OK for 2.6.16. But it looks as if the problem is not excessive CPU usage — so we're probably looking at a server problem, not a client problem here. Checking the Linux Kernel Mailing List show that the problem of RPC payload fragments not being page-aligned is a known problem, but the patch hasn't made it into mainline even though the issue was understood over a year ago.

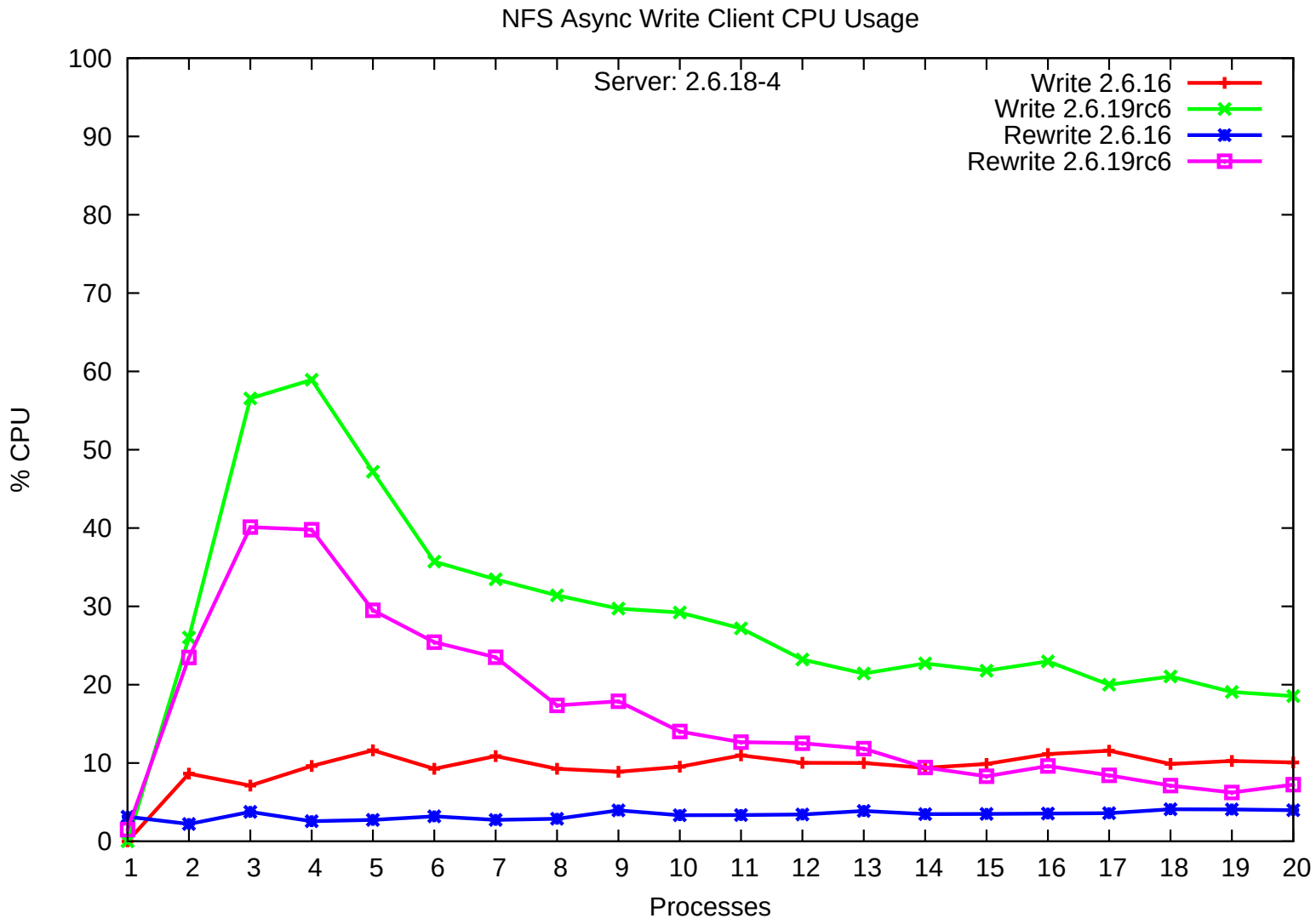
NFS Async Write Throughput



Asynchronous writes are around the same as synchronous writes.

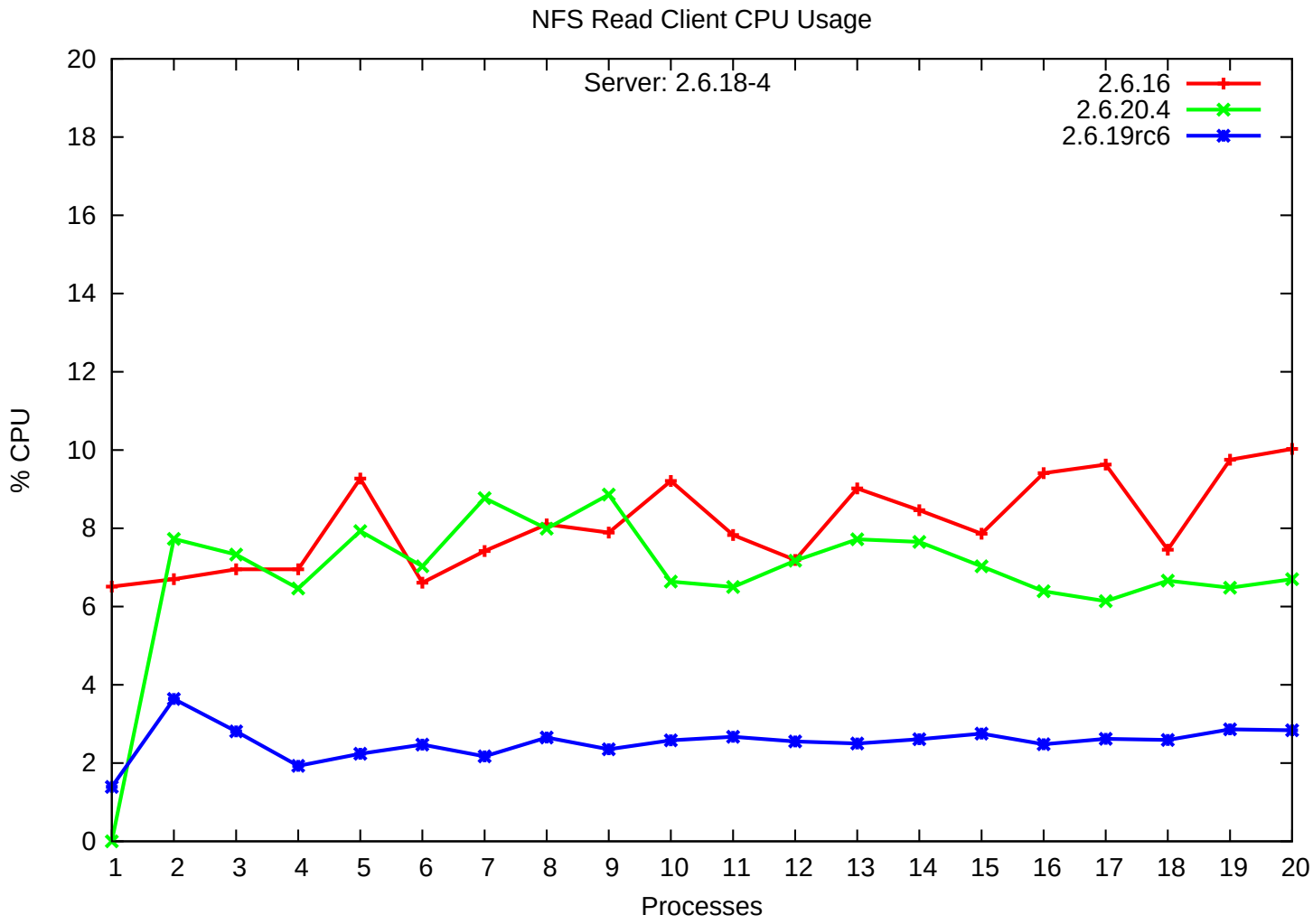
- Rewrite perf is still low
- Note Rewrite tput for Sync and Async mounts is almost similar.

NFS Async Client CPU usage



- More regression in Async mounted Rewrite.
- Rewrites have low perf due to:
  - High CPU usage at client
  - Fragments not page-aligned issue at the server
  - Fragments being written individually at server end.

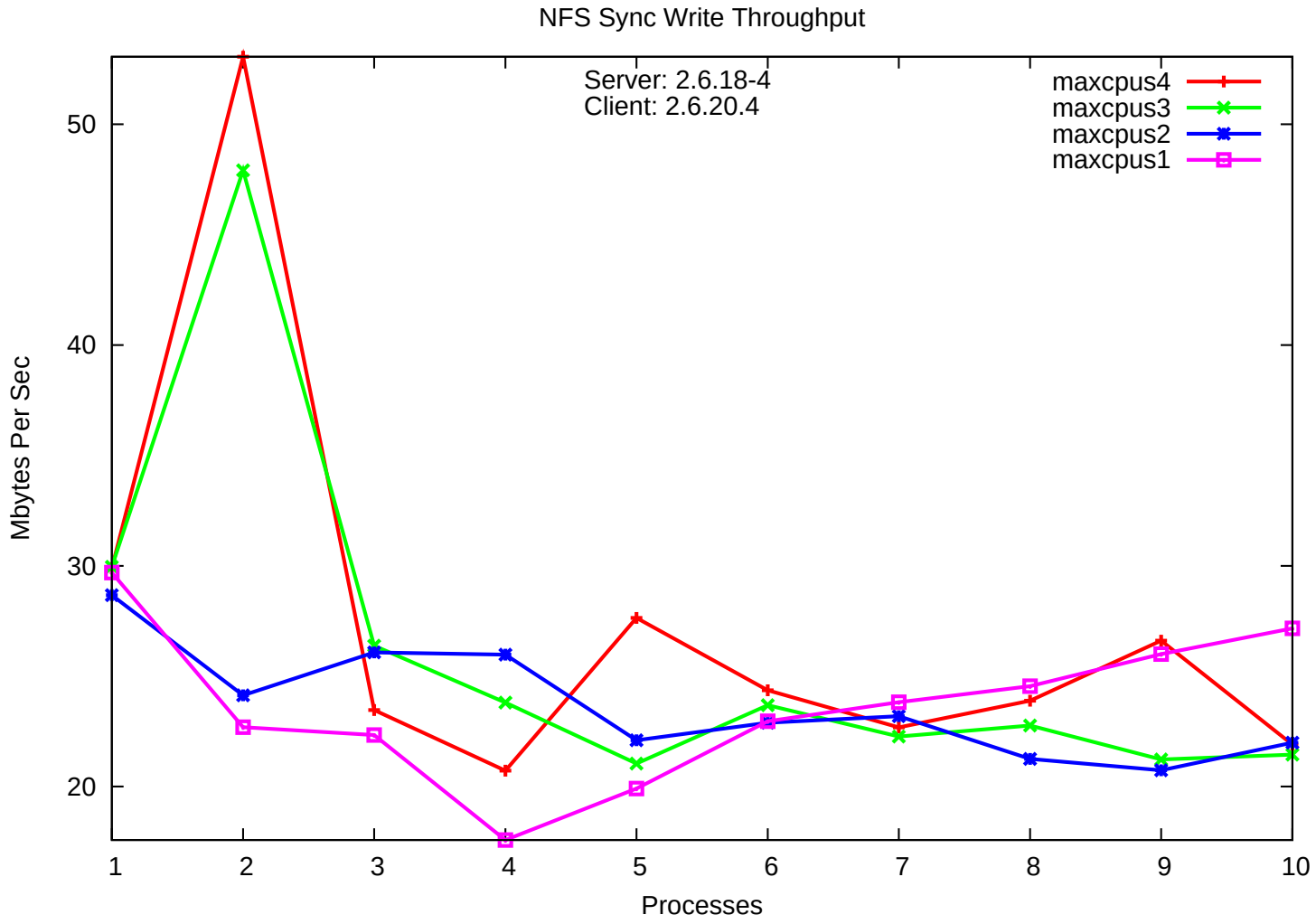
NFS Read client CPU usage



- CPU Usage for streaming read actually reduced between 2.6.16 and 2.6.19-rc6.

- But it’s back up again in 2.6.20.4

NFS Client Write Performance, varying CPUs



The curve is all over the place, with a major peak for two processes writing on three or four processors. There's an obvious contention problem here, probably (but not necessarily) on the Big Kernel Lock. We need to do more analysis to find out what's going on.

Benchmarking

Iozone Running Times

- For all tests in one go: Writes, Rewrites, Reads/Re-reads

Kernel Version	Duration
2.6.16	16 hours
2.6.19rc6	20 hours
2.6.20.4	35 hours

So you can see it all takes too long to be run routinely (yet).

The Issues

- NFS components span the stack from n/w and disk drivers to user space programs — Complex!
- Distributed file system theory and algorithms are at the core of it. – But also locking, caching and performance management

NFS is complex, spanning many parts of the Linux kernel and userspace
Consequently the maintainers are loathe to change things that may have bad interactions elsewhere. Also, we need more eyes.

What's next?

- Need co-ordinated effort to measure and report NFS perf.
- Ensure continuous perf. related feedback into Linux NFS dev channels
- Gelato@UNSW has one person working on this...not enough.
- Can we get a SIG together?

Can you be part of the effort?

Technical take-back

- Linux on Itanium at least works now . . .
- ..but it could fly . . .
- Finding out how, needs further investigation.

So that’s where we’re up to now. If you want to join in, talk to me.

Acknowledgements

HP provided servers and clients — And Money $$$

SGI provided an Altix machine

This work was funded by the Australian Research Council, HP, UNSW, and National ICT Australia.